Lecture Notes on Sequent Calculus

15-417/817: HOT Compilation
Frank Pfenning

Lecture 17
March 25, 2025

1 Introduction

From the perspective of proof theory, our focus has been on natural deduction (for the ND language) and the semi-axiomatic sequent calculus (for the Sax language). Hovering in the background has been the sequent calculus, one of the fundamental achievements of early proof theory [Gentzen, 1935]. Gentzen used it to establish consistency for first-order logic via cut elimination. Among his many insights were that classical and intuitionistic logic can be distinguished simply by whether more than one proposition is allowed in the succedent.

Many properties of logics are most cleanly expressed and proved via the sequent calculus, even if its computational interpretations have been somewhat slow in coming (see, for example, Ariola et al. [2009], Caires and Pfenning [2010]). If we want to know if a logic makes sense (and, indirectly, if a type system based on it makes sense) we usually fall back on the sequent calculus as our "gold standard". The key properties to check are cut and identity elimination. Their failure means that the rule system or the logic itself is likely to be flawed. Beyond that we also have focusing [Andreoli, 1992, Liang and Miller, 2009] as an almost universal property of logics derived from their sequent calculus formulation. We have seen part of the latter under the name of *inversion* in our development of nested pattern matching.

In today's lecture we take a look at the sequent calculus in the form where the structural rules that motivated adjoint logic are explicit. We start by considering an restriction of type-checking that is part of Lab 4.

2 Overloading

In an adjoint type system, a given definitions may have multiple different modes. These may have different implementations. For example, at a mode that does not allow contraction we may employ vertical reuse.

Because ND also supports subtyping for it equirecursive types, it may even have multiple different and incomparable types. We restrict such overloading to metavariables, that is, top-level definitions. How does this affect type checking?

Here is a simple example:

```
type bin[k] = +{'b0 : bin[k], 'b1 : bin[k], 'e : 1}
```

```
defn inc (x : bin) : bin = match x with
| 'b0(y) => 'b1(y)
| 'b1(y) => 'b0(inc y)
| 'e() => 'b1('e())
end

inst inc (x : bin[U]) : bin[U]
inst inc (x : bin[L]) : bin[L]
```

The increment function is overloaded in a straightforward way: a linear version (which should be able to reuse a lot of space) and a nonlinear version (which copies some bits in a number during an increment).

When checking the first instance, the recursive call to `inc` should also use the type unrestricted type. When checking the second instance, the recursive call should use use the linear type. In other words, the type checker has to *resolve overloading* by picking the correct instance of a definition.

The most straightforward technique for this purpose is simply trying each declaration in turn and backtracking as needed. In an expression such as `inc (inc (inc (inc x)))` this can be of exponential cost. At least concerning is the error message in case an expression is **not** type-correct. In a way, the best we can say is that all attempts at finding a type failed for a certain expression failed. Since we don't know which type was intended, this turns out to be rather unhelpful (as we experienced with the research compiler that performs just such backtracking).

One optimization is derived from looking at the rule

$$\frac{\Gamma \vdash s \Longrightarrow B \quad B \leq A}{\Gamma \vdash s \Longleftarrow A} \Rightarrow/\Leftarrow$$

Here, we have use the letter s (for *synthesizing*) to emphasize that the expression must synthesize. These are variables, instantiations of metavariables, and elimination form for negative types.

$$\text{Synthesizing Exps} \quad s \quad ::= \quad x \mid s\,e \mid s.k \mid s.\mathbf{force} \mid F[\sigma]$$

In general, it is expensive to synthesize a type for an expression, because in $s\,e$, the expression e must be checked against an argument type. Similarly, in $F[\sigma]$, the substitution σ must be checked against Δ from the definition $\mathbf{inst}\ F[\Delta] : B$. It is the latter case that introduces nondeterminism into the process, because there may be multiple instances for the same metavariables F.

Rather than backtracking over all these choices, we can quickly filter out those that have a chance to work in the larger context. That is, we change the $\Rightarrow/\Leftarrow$ rule to:

$$\frac{\Gamma \vdash s \gg B \quad B \leq A \quad \Gamma \vdash s \Longrightarrow B}{\Gamma \vdash s \Longleftarrow A} \Rightarrow/\Leftarrow$$

The first premise $\Gamma \vdash s \gg B$ is a quick check to identify those instances which **could** be subtypes of A. In the last premise we then only have to see if those candidates actually are correct.

The new judgment $\Gamma \vdash s \gg B$ synthesizes the *target type* B given Γ and s, without guaranteeing that s actually has the type. It is defined by the following rules, where we use Σ to refer to all

the instance declarations in a programs.

$$\frac{x : B \in \Gamma}{\Gamma \vdash x \gg B} \gg\mathsf{var} \qquad \frac{\Gamma \vdash s \gg A \to B}{\Gamma \vdash s\, e \gg B} \gg\!\to$$

$$\frac{\Gamma \vdash s \gg \&\{\ell : A_\ell\}_{\ell \in L} \quad (k \in L)}{\Gamma \vdash s.k \gg B_k} \gg\!\& \qquad \frac{\Gamma \vdash s \gg \uparrow B}{\Gamma \vdash s.\mathbf{force} \gg B} \gg\!\uparrow$$

$$\frac{\mathbf{inst}\ F\,[\Delta] : B \in \Sigma}{\Gamma \vdash F\,[\sigma] \gg B} \gg\mathsf{defn}$$

In addition to this optimization, we also need to make sure to check the definition of a metavariable against each of the given instance types.

At some level, the target type judgment can be used as a prefilter to cut down on nondeterminism and backtracking. Instead, in Lab 4 we explore the following option: pick the *lexically first instance* of a metavariable such that $\Gamma \vdash s \gg B$ for $B \leq A$ for this instance. This makes the type-checker **incomplete** because the first such instance may actually fail during full synthesis, while a later such instance may succeed.

Unfortunately, a second problem arises: synthesis is also used in the the general matching rule

$$\frac{\Gamma \vdash s \Longrightarrow B \quad \Delta \vdash B \rhd K^+ \Longleftarrow C}{\Gamma\,;\Delta \vdash \mathbf{match}\ s\ K^+ \Longleftarrow C}$$

Now we don't have a target to compare B to, and seeing which patterns may directly (or eventually) work is quite deterministic. The research compiler will just backtrack, where the scope of the backtracking is at least somewhat limited by the check against C.

For Lab 4, we require s not to contain a metavariable at its head so that the synthesized type is unique by the typing rules. This restriction is not really viable in practice, so an interesting research problem is how to allow some nondeterminism here without jeopardizing practical efficiency and while retaining reasonable error messages.

3 Comonads and Monads

Something else we have been meaning to mention is that the notion of comonad [Davies and Pfenning, 2001, Pfenning and Davies, 2001] and (strong) monad [Moggi, 1989, Wadler, 1992], both of which are significant in many lines of research on functional programming are easily derived in the adjoint framework.

We work with the following modes:

V	$\mathsf{V} > \mathsf{U}$	$\sigma(\mathsf{V}) = \{\mathsf{C}, \mathsf{W}\}$
U	$\mathsf{U} > \mathsf{X}, \mathsf{U} > \mathsf{L}$	$\sigma(\mathsf{U}) = \{\mathsf{C}, \mathsf{W}\}$
X		$\sigma(\mathsf{X}) = \{\mathsf{C}, \mathsf{W}\}$
L		$\sigma(\mathsf{L}) = \{\,\}$

and define

$!A$	$\triangleq$	$\downarrow^{\mathsf{U}}_{\mathsf{L}}\uparrow^{\mathsf{U}}_{\mathsf{L}} A_{\mathsf{L}}$	linear logic exponential
$\Box A$	$\triangleq$	$\downarrow^{\mathsf{V}}_{\mathsf{U}}\uparrow^{\mathsf{V}}_{\mathsf{U}} A_{\mathsf{U}}$	modal S4 necessity, a comonad
$\bigcirc A$	$\triangleq$	$\uparrow^{\mathsf{U}}_{\mathsf{X}}\downarrow^{\mathsf{U}}_{\mathsf{X}} A_{\mathsf{U}}$	lax modality, a strong monad

With these derived notions we can reanalyze or recreate the typical applications of monad and comonads in logic and programming.

4 Sequent Calculus

Now we return to our original lecture topic: the sequent calculus. We specify it in a form where the order of the antecedents is irrelevant, but other structural rules are explicit and only allowed based on the mode. We only given a brief introduction here; the full calculus and proofs of its properties can be found in Pruiksma et al. [2018].

Whenever we write a sequent $\Gamma \vdash A_m$, we presuppose that $\Gamma \geq m$, just as in its semi-axiomatic form. This *independence* is central to the proof of cut elimination.

We have the following structural rules.

$$\frac{(\mathsf{C} \in \sigma(m)) \quad \Gamma, A_m, A_m \vdash C_r}{\Gamma, A_m \vdash C_r}\ \mathsf{contract} \qquad \frac{(\mathsf{W} \in \sigma(m)) \quad \Gamma \vdash C_r}{\Gamma, A_m \vdash C_r}\ \mathsf{weaken}$$

The names of contraction and weakening make sense when reading these rules from the premises towards the conclusion. We also have the general rules of identity and cut.

$$\frac{}{A_m \vdash A_m}\ \mathsf{id} \qquad \frac{(\Gamma \geq m \geq r) \quad \Gamma \vdash A_m \quad \Delta, A_m \vdash C_r}{\Gamma, \Delta \vdash C_r}\ \mathsf{cut}$$

Notice that in the cut rule we do **not** use the context join operator $\Gamma\ ;\ \Delta$. Instead, we combine the antecedents from the two premises, possibly with duplicates. If their modes permit, these can then be contracted (using the rule contract).

Besides these general rules, each connective is then defined by its left and right rules, pertaining to an antecedent or the succedent, respectively. We show only two samples, referring to Pruiksma et al. [2018] for a complete set of rules.

$$\frac{\Gamma, A_m, B_m \vdash C_r}{\Gamma, A_m \otimes B_m \vdash C_r} \otimes L \qquad \frac{\Gamma \vdash A_m \quad \Delta \vdash B_m}{\Gamma, \Delta \vdash A_m \otimes B_m} \otimes R$$

$$\frac{(\Gamma \geq m) \quad \Gamma \vdash A_m \quad \Delta, B_m \vdash C_r}{\Gamma, \Delta, A_m \to B_m \vdash C_r} \to L \qquad \frac{\Gamma, A_m \vdash B_m}{\Gamma \vdash A_m \to B_m} \to R$$

In Sax, the right rules for positive connectives (here $\otimes R$) and the left rules for negative connectives (here $\to L$) turn into axioms. You can find the rules in Lecture 14.

The property of identity states that we need the identity rule only for atomic proposition A, and all others are derived. This is proved by an induction over the structure of A.

We also have cut elimination, that is, if we can prove a sequent $\Gamma \vdash A_m$ with cut, then there also is a proof without cut. This is usually proved by stating that the rule of cut is *admissible*, that is, if both premises can be derived without cut, so can the conclusion. We write this as

$$\frac{(\Gamma \geq m \geq r) \quad \Gamma \vdash A_m \quad \Delta, A_m \vdash C_r}{\Gamma, \Delta \vdash C_r}\ \mathsf{cut}$$

where the dashed line means that the rule is claimed to be admissible rather than primitive.

This is often proved by a nested induction, first on the structure of cut formula A_m and second simultaneously on the structure of the two premises (one must get smaller while the other stays the same). The rule of contraction throws a wrench into the works of such a proof, but it can be repaired by cutting multiple occurrences of the same formula A_m at once (see Gentzen [1935] for the original calculus, and Pruiksma et al. [2018] for the adjoint sequent calculus).

We show some key *cut reductions* because they are often at the core of the dynamics of a language. As an example, we can see that for the linear sequent calculus Caires and Pfenning [2010], Caires et al. [2016] where it corresponds to synchronous communication according to *session types* [Honda et al., 1998]. We show just two examples. In the first, we elide the independence constraints since they are trivially satisfied by presupposition.

$$\dfrac{\dfrac{\mathcal{D}_1}{\Gamma_1 \vdash A_m} \quad \dfrac{\mathcal{D}_2}{\Gamma_2 \vdash B_m}}{\Gamma_1, \Gamma_2 \vdash A_m \otimes B_m}\otimes R \quad \dfrac{\dfrac{\mathcal{E}'}{\Delta, A_m, B_m \vdash C_r}}{\Delta, A_m \otimes B_m \vdash C_r}\otimes L}{\Gamma_1, \Gamma_2, \Delta \vdash C_r}\ \mathsf{cut}_{A\otimes B}$$

$$\longrightarrow$$

$$\dfrac{\dfrac{\mathcal{D}_2}{\Gamma_2 \vdash B_m} \quad \dfrac{\dfrac{\mathcal{D}_1}{\Gamma_1 \vdash A_m} \quad \dfrac{\mathcal{E}'}{\Delta, A_m, B_m \vdash C_r}}{\Gamma_1, B_m, \Delta \vdash C_r}\ \mathsf{cut}_A}{\Gamma_1, \Gamma_2, \Delta \vdash C_r}\ \mathsf{cut}_B$$

We see that a cut at type $A \otimes B$ is replaced by two cuts, one at A and one at B, which means the induction measure becomes smaller.

As indicated above, contraction is more difficult.

$$\dfrac{(\Gamma \geq m \geq r) \quad \dfrac{\mathcal{D}}{\Gamma \vdash A_m} \quad \dfrac{(\mathsf{C} \in \sigma(m)) \quad \dfrac{\mathcal{E}'}{\Delta, A_m, A_m \vdash C_r}}{\Delta, A_m \vdash C_r}\ \mathsf{contract}}{\Gamma, \Delta \vdash C_r}\ \mathsf{cut}_A$$

$$\longrightarrow$$

$$\dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash A_m} \quad \dfrac{\dfrac{\mathcal{D}}{\Gamma \vdash A_m} \quad \dfrac{\mathcal{E}'}{\Delta, A_m, A_m \vdash C_r}}{\Gamma, \Delta, A_m \vdash C_r}\ \mathsf{cut}_A}{\dfrac{\Gamma, \Gamma, \Delta \vdash C_r}{\Gamma, \Delta \vdash C_r}\ \mathsf{contract} \times |\Gamma|}\ \mathsf{cut}_A$$

The reason we can apply contraction at the end of the derivation is that $\mathsf{C} \in \sigma(m)$ and $\Gamma \geq m$. By monotonicity, we also have $\mathsf{C} \in \sigma(k)$ for every $y : B_k \in \Gamma$.

The sad fact is, however, that while the top cut on $\mathcal{D}$ and $\mathcal{E}'$ is on smaller derivations, the second cut is not. So to turn this into a proof of the admissibility of cut, we need the idea of the multicut floated above (and found in Pruiksma et al. [2018]).

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

Zena M. Ariola, Aaron Bohannon, and Amr Sabry. Sequent calculi and abstract machines. *ACM Transactions on Programming Languages and Systems*, 31(4):13:1–13:48, May 2009.

Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. Special Issue on Behavioural Types.

Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *7th European Symposium on Programming Languages and Systems (ESOP 1998)*, pages 122–138. Springer LNCS 1381, 1998.

Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, November 2009.

Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Asilomar, California, June 1989. IEEE Computer Society Press.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

Klaas Pruiksma, Willow Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf.

Philip Wadler. The essence of functional programming. In *Conference Record of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, January 1992. ACM Press.